Звіт
Лабораторна робота 3
ОБ’ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ
Большаков Андрій МІТ-31
https://github.com/Utilka/OOP_labs_Univ

АБСТРАКТНІ КЛАСИ
Мета: ознайомитися з поняттям абстрактного класу, навчитися створювати абстрактні класи у Пайтон.

Створити абстрактний клас Шифратор , який встановлює параметри шифрування, зашифровує та розшифровує повідомлення. Створити два похідні класи: Симетричний_шифратор, Асиметричний_шифратор, Шифратор_Віжинера. Створити клас-контейнер, який вміщуватиме шифратори з різними параметрами.Створити абстрактний клас Шифратор , який встановлює параметри шифрування, зашифровує та розшифровує повідомлення. Створити два похідні класи: Симетричний_шифратор, Асиметричний_шифратор, Шифратор_Віжинера. Створити клас-контейнер, який вміщуватиме шифратори з різними параметрами.

```
import abc
from abc import ABCMeta

from random import randrange

import Crypto
from Crypto import Random
from Crypto.Random import get_random_bytes

from Crypto.Cipher import AES
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP


class MyCipher(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def encrypt(self, clear_message):
        pass

    @abc.abstractmethod
    def decrypt(self, encrypted_message):
        pass


class MySymmetric(MyCipher, metaclass=ABCMeta):

    def __init__(self, private_key):
        self._private_key = private_key


class MyAsymmetric(MyCipher, metaclass=ABCMeta):

    def __init__(self, private_key, public_key):
        self._public_key = public_key
        self._private_key = private_key
```

```python
class MyAES(MySymmetric):

    def __init__(self, private_key=None):
        if (private_key is None):
            private_key = self.generate_key()

        super().__init__(private_key)

    @staticmethod
    def generate_key(length=16):
        key = get_random_bytes(length)
        # key = b'Sixteen byte key'
        return key

    def encrypt(self, clear_message):
        cipher = AES.new(self._private_key, AES.MODE_EAX)
        nonce = cipher.nonce
        ciphertext, tag = cipher.encrypt_and_digest(clear_message)
        return (nonce, ciphertext, tag)

    def decrypt(self, encrypted_message):
        """

        :type encrypted_message: tuple (nonce,ciphertext,tag)
        """
        nonce, ciphertext, tag = encrypted_message

        key = b'Sixteen byte key'
        cipher = AES.new(self._private_key, AES.MODE_EAX, nonce=nonce)
        plaintext = cipher.decrypt(ciphertext)
        try:
            cipher.verify(tag)
            return plaintext
        except ValueError:
            print("Key incorrect or message corrupted")


class MyRSA(MyAsymmetric):

    def __init__(self, private_key=None, public_key=None):
        if (private_key is None):
            private_key = self.generate_key()
        if (public_key is None):
            public_key = private_key.publickey()

        super().__init__(private_key, public_key)

    @staticmethod
    def generate_key(length=4096):
        private_key = RSA.generate(length)
        public_key = private_key.publickey()
        return private_key

    def encrypt(self, clear_message):
        encryptor = PKCS1_OAEP.new(self._public_key)
        encrypted = encryptor.encrypt(clear_message)
        return encrypted

    def decrypt(self, encrypted_message):
        decrypter = PKCS1_OAEP.new(self._private_key)
```

```
        decrypted = decrypter.decrypt(encrypted_message)
        return decrypted


class MyVigenere(MySymmetric):
    def __init__(self, private_key=None):
        if (private_key is None):
            private_key = self.generate_key()

        super().__init__(private_key)

    @staticmethod
    def generate_key(length=10):
        private_key = "".join([chr(randrange(26) + 65) for i in range(length)])
        return private_key

    def _v_cypher(self, text, enc):
        key = self._private_key
        if isinstance(text, bytes):
            text = text.decode("utf-8")
        text = text.upper()
        key_length = len(key)
        key_as_int = [ord(i) for i in key]
        text_as_int = [ord(i) for i in text]
        if enc:
            ciphertext = ''
            for i in range(len(text_as_int)):
                value = (text_as_int[i] + key_as_int[i % key_length]) % 26
                ciphertext += chr(value + 65)
            return ciphertext
        else:
            plaintext = ''
            for i in range(len(text_as_int)):
                value = (text_as_int[i] - key_as_int[i % key_length]) % 26
                plaintext += chr(value + 65)
            return plaintext

    def encrypt(self, plaintext):
        return self._v_cypher(plaintext, True)

    def decrypt(self, ciphertext):
        return self._v_cypher(ciphertext, False)


if __name__ == '__main__':
    my_rsa = MyAES()
    enc_m = my_rsa.encrypt(b"someMyAEStext")
    dec_m = my_rsa.decrypt(enc_m)
    print(dec_m)

    my_rsa = MyRSA()
    enc_m = my_rsa.encrypt(b"someMyRSAtext")
    dec_m = my_rsa.decrypt(enc_m)
    print(dec_m)

    my_vig = MyVigenere()
    enc_m = my_vig.encrypt("someVigenetext")
    dec_m = my_vig.decrypt(enc_m)
    print(dec_m)

import abc
from abc import ABCMeta
```

```python
from random import randrange

import Crypto
from Crypto import Random
from Crypto.Random import get_random_bytes

from Crypto.Cipher import AES
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP


class MyCipher(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def encrypt(self, clear_message):
        pass

    @abc.abstractmethod
    def decrypt(self, encrypted_message):
        pass


class MySymmetric(MyCipher, metaclass=ABCMeta):

    def __init__(self, private_key):
        self._private_key = private_key


class MyAsymmetric(MyCipher, metaclass=ABCMeta):

    def __init__(self, private_key, public_key):
        self._public_key = public_key
        self._private_key = private_key


class MyAES(MySymmetric):

    def __init__(self, private_key=None):
        if (private_key is None):
            private_key = self.generate_key()

        super().__init__(private_key)

    @staticmethod
    def generate_key(length=16):
        key = get_random_bytes(length)
        # key = b'Sixteen byte key'
        return key

    def encrypt(self, clear_message):
        cipher = AES.new(self._private_key, AES.MODE_EAX)
        nonce = cipher.nonce
        ciphertext, tag = cipher.encrypt_and_digest(clear_message)
        return (nonce, ciphertext, tag)

    def decrypt(self, encrypted_message):
        """

        :type encrypted_message: tuple (nonce,ciphertext,tag)
        """
        nonce, ciphertext, tag = encrypted_message
```

```
        key = b'Sixteen byte key'
        cipher = AES.new(self._private_key, AES.MODE_EAX, nonce=nonce)
        plaintext = cipher.decrypt(ciphertext)
        try:
            cipher.verify(tag)
            return plaintext
        except ValueError:
            print("Key incorrect or message corrupted")


class MyRSA(MyAsymmetric):

    def __init__(self, private_key=None, public_key=None):
        if (private_key is None):
            private_key = self.generate_key()
        if (public_key is None):
            public_key = private_key.publickey()

        super().__init__(private_key, public_key)

    @staticmethod
    def generate_key(length=4096):
        private_key = RSA.generate(length)
        public_key = private_key.publickey()
        return private_key

    def encrypt(self, clear_message):
        encryptor = PKCS1_OAEP.new(self._public_key)
        encrypted = encryptor.encrypt(clear_message)
        return encrypted

    def decrypt(self, encrypted_message):
        decrypter = PKCS1_OAEP.new(self._private_key)
        decrypted = decrypter.decrypt(encrypted_message)
        return decrypted


class MyVigenere(MySymmetric):
    def __init__(self, private_key=None):
        if (private_key is None):
            private_key = self.generate_key()

        super().__init__(private_key)

    @staticmethod
    def generate_key(length=10):
        private_key = "".join([chr(randrange(26) + 65) for i in range(length)])
        return private_key

    def _v_cypher(self, text, enc):
        key = self._private_key
        if isinstance(text, bytes):
            text = text.decode("utf-8")
        text = text.upper()
        key_length = len(key)
        key_as_int = [ord(i) for i in key]
        text_as_int = [ord(i) for i in text]
        if enc:
            ciphertext = ''
            for i in range(len(text_as_int)):
                value = (text_as_int[i] + key_as_int[i % key_length]) % 26
```

```
            ciphertext += chr(value + 65)
        return ciphertext
    else:
        plaintext = ''
        for i in range(len(text_as_int)):
            value = (text_as_int[i] - key_as_int[i % key_length]) % 26
            plaintext += chr(value + 65)
        return plaintext

def encrypt(self, plaintext):
    return self._v_cypher(plaintext, True)

def decrypt(self, ciphertext):
    return self._v_cypher(ciphertext, False)


if __name__ == '__main__':
    my_rsa = MyAES()
    enc_m = my_rsa.encrypt(b"someMyAEStext")
    dec_m = my_rsa.decrypt(enc_m)
    print(dec_m)

    my_rsa = MyRSA()
    enc_m = my_rsa.encrypt(b"someMyRSAtext")
    dec_m = my_rsa.decrypt(enc_m)
    print(dec_m)

    my_vig = MyVigenere()
    enc_m = my_vig.encrypt("someVigenetext")
    dec_m = my_vig.decrypt(enc_m)
    print(dec_m)
```

```
from collections.abc import Collection
from main import MyCipher


class MyCipherColl(Collection):

    def __init__(self, cypher_list):
        for item in cypher_list:
            if not isinstance(item, MyCipher):
                raise TypeError("all objects in collection must be of type MyCipher")
        self._list = list(cypher_list)
        self.__index = 0

    def __contains__(self, item):
        return (item in self._list)

    def __iter__(self):
         # self.__index = 0
        return self._list.__iter__()

    # def __next__(self):
    #     if self.__index == len(self._list):
    #         raise StopIteration
    #     x = self._list[self.__index]
    #     self.__index += 1
```

```
    #     return x

    def __len__(self):
        return len(self._list)

if __name__ == '__main__':
    import main
    c= MyCipherColl([main.MyAES(),main.MyRSA(),main.MyVigenere(),main.MyVigenere()])
    for i in c:
        enc=i.encrypt(b"Mytext")
        dec=i.decrypt(enc)
        print(enc)
        print(dec)

    for i in c:
        enc = i.encrypt(b"Mytext")
        dec = i.decrypt(enc)
        print(enc)
        print(dec)
```

Висновок: я ознайомився з поняттям абстрактного класу, навчився створювати абстрактні класи у Пайтон.